Naval Research Laboratory
Washington, DC 20375-5320

NRL/MR/5540--94-7640

Design Documentation for the SINTRA Preprocessor

MYONG H. KANG

Center for Computer High Assurance Systems
Information Technology Division

RODNEY PEYTON

Kaman Sciences Corporation
Alexandria, VA

December 12, 1994

Approved for public release; distribution unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave Blank*)

2. REPORT DATE

December 12, 1994

3. REPORT TYPE AND DATES COVERED

1993

4. TITLE AND SUBTITLE

Design Documentation for the SINTRA Preprocessor

5. FUNDING NUMBERS

PE - 0303401G

6. AUTHOR(S)

Myong H. Kang and Rodney Peyton*

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/5540--94-7640

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Security Agency
9800 Savage Road
Ft. George G. Meade, MD 20755

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

*Kaman Sciences Corporation, Alexandria, VA 22303

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

In this document, we present the detailed description of the Secure INformation Through Replicated Architecture (SINTRA) preprocessor. The SINTRA preprocessor is an implementation of the replicated data mode [CKF94]. We describe the role of the preprocessor in the SINTRA system, the internal code structure, and high level specifications of the preprocessor.

We have prepared this report for system designers and programmers who want to understand the structure of the SINTRA preprocessor. We also hope this report is also helpful to the people who will maintain the SINTRA preprocessor code. In this report, we assume that the reader is familiar with the material presented in [CKF94] and [Kan94].

14. SUBJECT TERMS

Database
Compiler

15. NUMBER OF PAGES

24

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT

UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT

UNCLASSIFIED

20. LIMITATION OF ABSTRACT

UL

CONTENTS

Design Documentation for the SINTRA Preprocessor

1 Introduction

The SINTRA database system is a multilevel trusted database management system based on the replicated architecture [FrM89, Kan94]. The replicated architecture approach uses a physically distinct backend database management system for each security level. Each backend database contains information at a given security level and all data from lower security levels. The system security is assured by a trusted frontend which permits a user to access only the backend database system which matches his/her security level.

The SINTRA database system consists of one trusted front end (TFE), several untrusted backend database systems (UBD) and several User Interface Stations (UIS). The role of the TFE includes user authentication, directing user queries to the backend, maintaining data consistency among backends, etc. Each UBD can be any commercial off-the-shelf database system and each UIS can be any system supporting Unix, X11 and TCP/IP.

1.1 The SINTRA Database System

The SINTRA database system, which is currently being prototyped at the Naval Research Laboratory, uses the HFSI XTS-300 system as a trusted frontend and untrusted ORACLE DBMSs which are running on SUN4/300 as backend databases. The backend and frontend computers are connected through Ethernet. Figure 1 illustrates the SINTRA architecture where NI represents the network interface process (for more detailed diagram, see [Kan94, Fig3]). There are two components between the trusted frontend and an off-the-shelf database: (1) a global scheduler and (2) a query preprocessor. These two components perform the systems transaction management functions and assure the consistency and integrity of replicated data among different backend databases. Notice that the global scheduler has a portion resident in the trusted frontend and another portion in each (untrusted) backend. Each ORACLE DBMS has a local scheduler.

Before the responsibilities of the global scheduler are discussed, we define two classes of transactions:

Definition 1. A **user transaction** T_i is a sequence of queries terminated by either a *commit(c_i)* or an *abort(a_i)*, i.e., T_i = <q_{i1}, q_{i2}, ..., q_{in}, c_i>. Each query, q_{ij}, is an atomic operation and is one of `retrieve`, `insert`, `replace`, or `delete`.

Once a user transaction is successfully committed and that transaction contains an

Manuscript approved September 22, 1994.

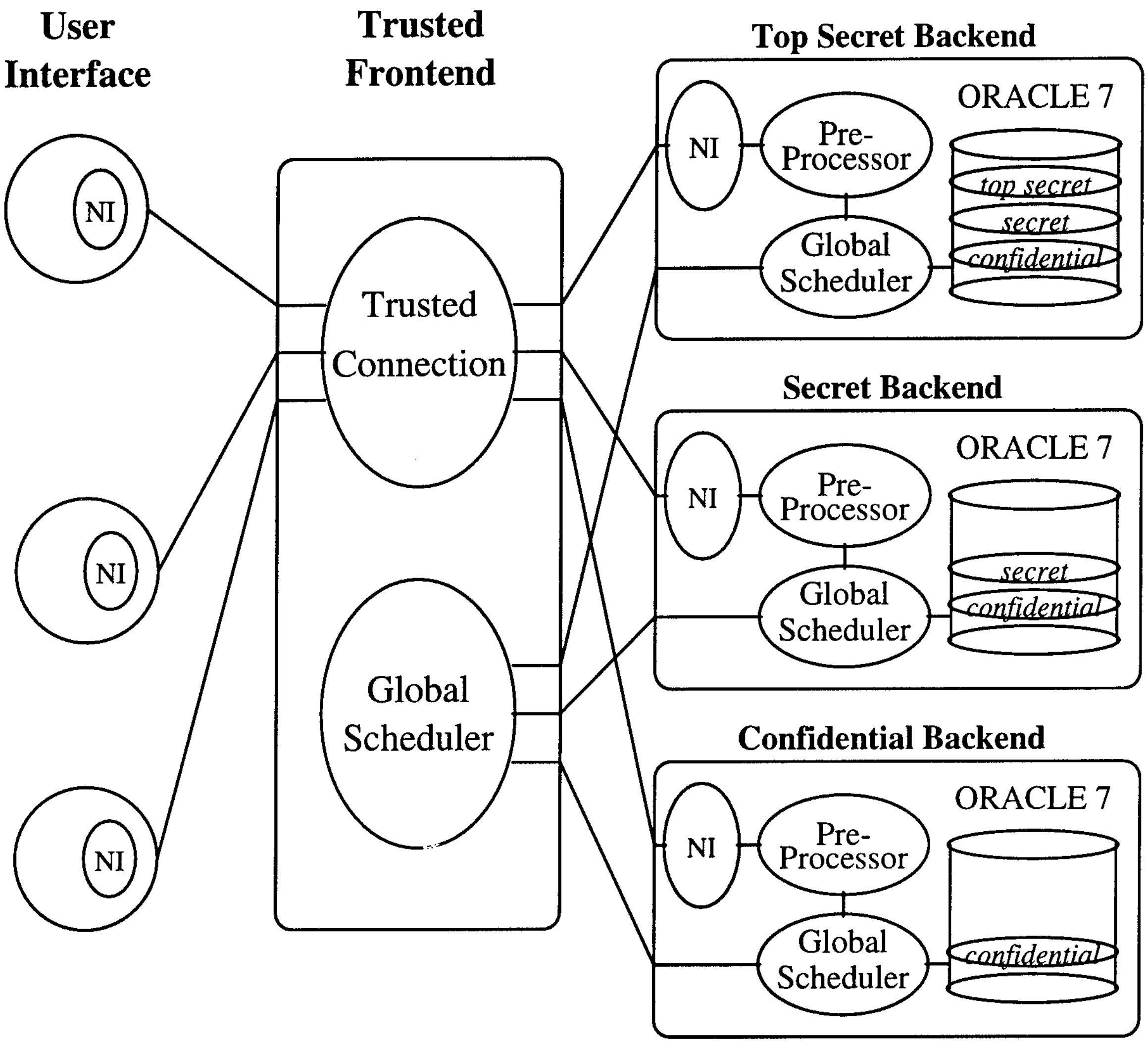

Figure 1: The SINTRA Architecture.

update query, then the corresponding *update projection* is spawned and propagated to upper levels to modify the replicas located at higher security level backends.

Definition 2. An **update projection** U_i, which corresponds to a transaction T_i, is a sequence of update queries, e.g., U_i = $<q_{i2}$, q_{i5}, ... q_{in}, $c_i>$ obtained from transaction T_i by simply removing all `retrieve` queries.

The global scheduler performs the following tasks:

- Receive queries from the preprocessor and the global scheduler at lower security levels and send them to the appropriate backend database.
- When a transaction is committed, send the corresponding update projection to

higher security level backends so that the consistency among replicas is maintained.

A transaction model and a global scheduler for the SINTRA database system have been presented in [KFC93, KaP93].

1.2 The Role of The Query Preprocessor

The SINTRA query preprocessor plays an important role in maintaining data consistency among different backend databases, preserving data integrity, and bridging the semantic gap between conventional and multilevel-secure databases. The SINTRA query preprocessor modifies user queries based on the replicated architecture data model, the replicated architecture relational algebra, and the semantics of the replicated architecture update operations described in [CKF94].

The following are responsibilities of the SINTRA query preprocessor:

1. If a secret-level user were allowed to modify the copy of a confidential data item in the secret-level backend database, then inconsistent database states between the secret and confidential backend databases could be created (assuming no *write-down* is allowed). To prevent such inconsistencies, the query preprocessor must inspect and modify each user's update queries so that the backend database system only modifies the secret-level data items (i.e., whose tuple level classification is the same as the user's login level) - it is also assumed that no *write-up* is allowed. Notice that this behavior enforces integrity and consistency among the backend databases. Confidentiality is enforced by preventing write-down and by limiting each user's queries to the backend corresponding to his login levels.

2. There is also some data that can be disclosed but cannot be modified by the user – it can be modified only by the system. For instance, the classification of a tuple cannot be modified by the user. It is the responsibility of the query preprocessor to guarantee the integrity of security label data.

3. SINTRA, which is a multilevel relational database system, uses conventional relational database systems as backend databases. These conventional relational databases use SQL, which is based on the conventional (single-level) relational algebra and the semantics of conventional update operations [Ull82]. On the other hand, the whole multilevel relational database is based on a multilevel relational algebra and the semantics of multilevel relational update operations which were presented in [CKF94]. Therefore, before SINTRA user queries, which are posed to

the MLS database, are submitted to the backend databases, they must be translated into other queries, which are based on the conventional relational algebra and the semantics of conventional update operations.

4. Since there are semantic differences between user queries and the queries passed to the backend database, a single user query may be translated into several queries for the backend database. In such cases, the query preprocessor has to guarantee the atomicity of each user query. For example, if a single update query is translated into three queries for the backend database, then this sequence of three queries has to be executed as a single atomic action i.e., it is submitted to the backend as a single subtransaction.

5. In the SINTRA system, the user view of a relation (table) may not be the same as the relation stored in the backend database. The query preprocessor has to modify user queries so that implementation detail can be hidden from the user.

To perform the above responsibilities, the SINTRA query preprocessor at the appropriate backend intercepts, inspects and modifies user queries before they are submitted to the ORACLE DBMS.

Note that only the original user transactions have to be modified by the preprocessor; the queries in update projections bypass the preprocessor (because those have been modified already at a lower level).

2 Overview of Code Structure

Figure 2 illustrates the internal process structure of the SINTRA query preprocessor. Once a user query is submitted to the SINTRA system, it will be passed to the query preprocessor (i.e., process1). The output of the final process of the query preprocessor will be passed to the ORACLE DBMS. The function of each process is as follows:

1. YACC++ and LEX++ are used to parse user queries and convert them to an internal representation (IR). Parse trees are used as the internal representation.

2. This process is responsible for carrying out responsibilities (1) and (2) in section 1.2. Each user query is inspected so that only legal queries are passed to the next step. For example, if a high-user query tries to delete the replicas of low level tuples located at the high backend, the validation routine will reject that query.

3. This process is responsible for carrying out responsibilities (3) and (4) in section 1.2. Parse trees are modified based on a multilevel relational algebra and the semantics of multilevel relational update operations.

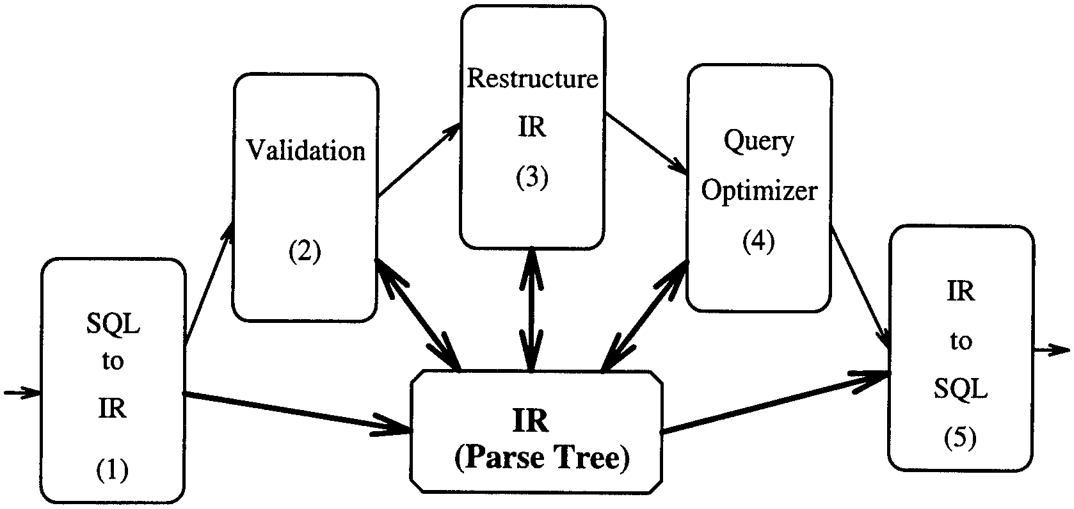

Figure 2: Internal Process Structure of Query Preprocessor.

4. Query optimization is performed based on intimate knowledge of query preprocessor implementation. A detailed description of the SINTRA query optimizer will appear in [Kang].
5. The process (5) converts the internal representation (parse tree) to SQL and submits it to ORACLE.

The individual processes are separated so that each process can be developed, tested, and upgraded or replaced more easily.

All the SINTRA preprocessor codes are written in C++, an object-oriented programming language. We do not consider the design methodology of the SINTRA preprocessor to be objected-oriented, but the *dynamic binding* feature of C++ provided us the flexibility to navigate the parse tree easily without explicit testing of classes.

3 Organization of Relations

There are two straight forward ways to organize relations at the backend database in the SINTRA architecture. We discuss each method and its advantages and disadvantages. We do not consider the *SeaView* style decomposition [Den87] because that method requires many join operations to reconstruct a multilevel relation.

Option A

A relation is organized as the user views it. For example, relation R will be organized as:

$R(A_1,\ C_1,\ A_2,\ C_2,\ .\ .\ .,\ A_n,\ C_n,\ TL)$

where TL is the tuple classification level, and it can range from system low to system high.

One advantage of option A is that there is no need to create views and perform unions as in option B. Therefore, it is more efficient in some cases.

Option B

A relation is decomposed as several base relations – one base relation for each security class. For example, relation R in the high backend database may be divided into R_H and R_L. Relations R_H and R_L will be organized as:

$R_H(A_1,\ C_1,\ A_2, C_2,\ .\ .\ .,\ A_n,\ C_n)$

$R_L(A_1,\ C_1,\ A_2, C_2,\ .\ .\ .,\ A_n,\ C_n)$

There are several advantages of option B over option A.

- It provides better concurrency. Some off-the-shelf databases prohibit simultaneous update operations on the same relation. The organization of option B enables simultaneous update operations on relation R – high users can update R_H while update projections from the low level can modify R_L.
- The join operation is much easier under option B when the security classes of a system do not form a completely ordered set. In this case, tuples from incomparable security classes should not participate in the join, and option B simplifies their segregation.
- It reduces work in some cases. For instance, a high-level user query `delete from R where` *cond* may be translated into `delete from` R_H `where` *cond* rather than `delete from R where` *cond* `and TL = 'H'` by the query preprocessor.

Even though the organization of option B may complicate the query preprocessor in some cases, we prefer option B in the SINTRA prototype because

1. we do not want a user transaction and an update projection to interfere with

each other, and under option B, these would operate on separate base relations [KFC93].

2. We also believe that the data organization of option B provides more opportunity for query optimization and parallelization. Consider a H-user query `delete from R where R.a = S.d and S.TL = 'H'`. The output of the restructured and optimized query will be `delete from` R$_H$ `where` R$_H$`.a =` S$_H$`.d`. The general query optimization and parallelization issues are beyond the scope of this document but will be addressed in [Kang].

4 Implementation Specifications

Based on the structure of relations and the replicated data model [CKF94], the SINTRA query preprocessor translates user queries to queries that are ready to be submitted to ORACLE. The SINTRA preprocessor enforces the following rules:

1. T > S > C is the SINTRA security structure.
2. Character '#' is a preprocessor reserved character unless it appears in literals (e.g., 'JACK#').
3. User can use attribute names which contain '#' character only in `where` clause.
4. User cannot use table names which contain '#' character.
5. Before each user query is translated, `savepoint` is inserted.
6. There are global variables SL (user's session level) and Role which will be pre-set by the system.

In the following sections, we give examples that show how each type of query is translated. The following database schemata are used in our examples. We also call the user whose session level is X an X-user.

R(A, A#, B, B#, C, C#, T#L)

S(D, D#, E, E#, F, F#, T#L)

T(G, G#, H, H#, I, I#, T#L)

where *A#* is the classification of an attribute A and *T#L* is a tuple-level classification.

4.1 Create Tables

In the SINTRA system, only the database administrator (DBA) can issue the create table command. We recommend that a DBA issues create table at the lowest level; it will be propagated to higher security levels. However, unlike other update operations, the create command will be propagated unmodified, and it will pass through the preprocessor at each level.

```
create table R (
  A char(10),
  A# char(1),
  B number(5),
  B# char(1),
   . . . )
```

is an illegal query because A# and B# are used by the user. However,

```
create table R (
  A char(10),
  B number(5),
  C char (9) );
```

is legal, and the preprocessor will produce the following statements:

```
create table R#C (
  A char(10),
  A# char(1),
  B number(5),
  B# char(1),
  C char (9),
  C# char(1) );

create view R (A, A#, B, B#, C, C#, T#L)
as select A, A#, B, B#, C, C#, 'C'
   from R#C;
```

When the create table command is propagated to the 'S' level, it will generate the following statements.

```
create table R#C (
    . . . )

create table R#S (
    . . . )

create view R (A, A#, B, B#, C, C#, T#L)
as select A, A#, B, B#, C, C#, 'S'
   from R#S
   UNION
   select A, A#, B, B#, C, C#, 'C'
   from R#C;
```

When the `create table` command is propagated to the T-backend, it will generate three relations, R#C, R#S, and R#T, as well as a view that combines these three relations. In this document, we refer R#C, R#S, and R#T as base relations.

Since all relations in SINTRA are owned by the DBA[2], the DBA should grant access for updating and retrieving data to proper users.

4.2 Complex Conditions

Each query has a qualification clause (`where` conditions) that specifies which tuples of the relation are to be retrieved or updated. We distinguish two kinds of qualification clauses: simple and complex. A complex qualification requires examination of multiple relations in order to determine affected set of tuples, whereas a simple qualification can be evaluated by examining only one relation. Each qualification clause is a Boolean combination of atomic conditions. Each atomic condition may specify a selection or a join operation, and each condition is connected by connectives.

Each atomic condition, in turn, may be either a simple or complex predicate. A complex predicate that contains more than one relation will be interpreted as follows:

R.a @ S.d will be translated as (R.a @ S.d and R.a# = S.d#)

where @ = $\{ =, \geq, \leq, >, < \}$. R.a != S.d will be translated as (R.a != S.d or R.a# != S.d#).

[2] In this prototype, the DBA owns all relations because the consistency of database schema throughout different security levels, the uniqueness of relation name, and granting access to replicas of relations can be easily controlled by the DBA.

4.3 Select

select statement is relatively easy to translate. The basic ideas that we use in our translation are:

- Unsophisticated user's view of a multilevel relation is $R(A_1, A_2, \ldots, A_n, TL)$ and the information on data classification will always be provided.
- A user does not need to know the basic structure of the SINTRA relations. All translation from the SINTRA created view to base relations should be automatic.

```
select A, A#, B, B#, T#L
from R#T
```

is an illegal user statement because '#' character appears in from statement. Also A#, B#, and T#L cannot be specified by the user except where clause.

However, the following user statement is legal:

```
select A, B, C
from R
where T#L = 'T';
```

and will yield the following statement from the preprocessor:

```
SELECT A, A#, B, B#, C, C#, R.T#L
   FROM SINTRA.R
   WHERE T#L = 'T';
```

A user query:

```
select *
from R
where R.a > R.b;
```

will produce the following query:

```
SELECT A, A#, B, B#, C, C#, R.T#L
   FROM SINTRA.R
   WHERE SINTRA.R.A > SINTRA.R.B;
```

To obtain column (attribute) information, the SINTRA preprocessor issues the following query:

```
select column_name
from COLS
where table_name = 'R';
```

The following user query

```
select r.* from R, S
where R.a = S.d;
```

will produce

```
SELECT R.A, R.A#, R.B, R.B#, R.C, R.C#, R.T#L
   FROM SINTRA.R, SINTRA.S
   WHERE (SINTRA.R.A = SINTRA.S.A AND SINTRA.R.A# = SINTRA.S.D#);
```

4.4 Insert

We distinguish two types of insert queries: simple and complex. The difference between simple and complex insert queries are: the values to be inserted are specified via the VALUE argument in the simple insert query, whereas the values to be inserted in a complex insert query are specified via a subquery statement.

Simple Insert

```
insert into R (A, A#, B, B#)
 . . .
```

is an illegal user statement because '#' character appears. However, the following T-user's query is legal:

```
insert into R (A, B)
values ('Iowa', 'Battle ship');
```

and the preprocessor will produce the following statement:

```
insert into SINTRA.R#T (A, A#, B, B#, C, C#)
values ('Iowa', 'T', 'Battle ship', 'T', null, 'T');
```

Note that if there are more attributes in the relation than what user specified, then *NULL* and session level will be padded.

Complex Insert

An S-user's query:

```
insert into R (A, B, C)
select S.d, S.e, S.f
from S
where S.d > 5;
```

will produce

```
insert into SINTRA.R#S (A, A#, B, B#, C, C#)
select S.d, S.d#, S.e, S.e#, S.f, S.f#
from SINTRA.S
where SINTRA.S.d > 5;
```

Note that the elimination of duplicate tuples is the responsibility of the underlying DBMS.

4.5 Update

The treatment of the update statement is little more complex due to polyinstantiation. The following query:

```
update R
set A = 'Iowa', A# = 'T'
where . . .
```

is an illegal query because '#' character appears in `set` clause.

However, the following T-user's query:

```
update R
set A = ’Iowa#’
where C = ’Battle ship’;
```

is a legal query. To perform the polyinstantiation, the SINTRA preprocessor has to know the key attributes of the relation which will be updated. The preprocessor will search the primary key of relation R from the ORACLE metadata using the following command:

```
select COLUMN_NAME from user_ind_columns, user_constraints
where user_ind_columns.table_name = ’R’ and
user_constraints.table_name = ’R’ and
and user_constraints.constraint_type = ’P’ and
user_ind_columns.INDEX_NAME = user_constraints.constraint_name;
```

If the above query returns empty (i.e., the primary key was not defined) then all attribute names will be the primary key.

Once the primary keys are found then it will produce the following sequence of queries:

```
create table temp as
(select * from SINTRA.R#S        ; polyinstantiated tuples
 where SINTRA.R#S.C = ’Battle ship’
 union
 select * from SINTRA.R#C
 where SINTRA.R#C.C = ’Battle ship’ );

update temp
set A = ’Iowa#’,  A# = ’T’
where C = ’Battle ship’;

delete from SINTRA.R#T where <list of real keys> in
(select <list of real keys> from temp);

update SINTRA.R#T
set A = ’Iowa#’, A# = ’T’
where SINTRA.R#T.C = ’Battle ship’;
```

```
insert into SINTRA.R#T
(select * from temp);

drop table temp;
```

where a *list of real keys* includes user defined key and all classification attributes.

Note that if the default configuration of ORACLE (which does not produce serializable schedules) is used, the base relations which form the `temp` relation have to be locked explicitly to force the conflict among update queries. The temporary relation name should be carefully chosen so that no other `update` query can create the same temporary relation. The SINTRA preprocessor actually uses `t#emp`n as temporary relation names where n is a unique sequence number generated by ORACLE.

4.6 Delete

The following T-user's query:

```
delete from R
where R.B < 300 and B# = 'S';
```

is a legal user statement because '#' character appears in `where` clause. The preprocessor will produce the following statement:

```
delete from SINTRA.R#T
where SINTRA.R#T.B < 300 and SINTRA.R#T.B# = 'S';
```

However, the following is an illegal statement:

```
delete from R
where R#T.B < 300 and B# = 'S';
```

because users should not know that there is a base relation whose name is R#T.

4.7 Grant

The `grant` command will not be propagated to higher levels. Hence, if one wants to grant access to objects, he/she has to do it separately at each level.

For example, S-user's command:

```
grant all on R to public;
```

will be translated as

```
grant all on SINTRA.R#C to public;
grant all on SINTRA.R#S to public;
grant all on R to public;
```

We decided not to propagate the `grant` statement because:

- the user who is granted access to certain relations at one security level may not have access rights to the replica of the same relation at higher security level backends, and
- since all update projections are owned by the DBA it is not a problem to execute update projections at higher security levels.

4.8 Commit and Rollback

The `commit` and `rollback` commands will have their usual meaning. If a user transaction is successfully committed and the transaction contains update queries, then a corresponding update projection will be spawned and propagate to higher security levels.

5 Internal Representation

Every SQL statement is represented internally as a parse tree. Non-leaf nodes represent non-terminal grammar symbols and leaf nodes represent terminal symbols. Every node within the parse tree has a corresponding type and the type is represented in C++ as class definition. There is a base type defined named `parse_nd` which provides most of the methods needed to support the semantics required by each node within the parse tree. `Parse_nd` is inherited by the other types. All the methods defined in parse_nd are defined as virtual methods to allow the exploitation of the dynamic binding capabilities available in C++. Through the use of dynamic binding, in most cases, the required restructuring can be performed on a node without checking the type of the node or the type of the node's parent. If the semantics required by a node are supported by a method in class parse_nd then that method is executed via inheritance, otherwise, dynamic binding is

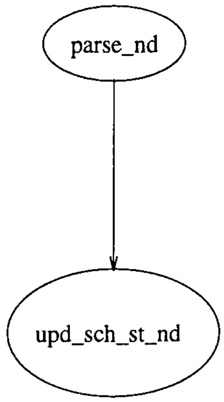

Figure 3: Typical Inheritance within the Preprocessor

used to access the appropriate method in the derived class to perform the required actions.

Figure 3 shows the inheritance relationship between a `parse_nd` and an update statement node. This type of inheritance (one level) is the only type used throughout the preprocessor to represent the parse tree.

5.1 Preprocessor Algorithm

1. After receiving a SQL statement from standard input, parse it using lex++ and yacc++. Yacc++ builds a parse tree using user defined types similar to the one defined in Figure 3.

2. Starting with the root of the parse tree returned by step 1, visit each node validating its syntax against SINTRA's supported subset of SQL and its semantics against SINTRA's security policy. This step is referred to as the validation phase. If the validation step fails, further parsing of the statement is aborted and the parser returns to step 1. For example, `delete from R where A# = 'C'` by an *S*-user will fail at this stage because the SINTRA security policy does not allow an *S*-user to insert *C*-data.

3. Starting with the root of the parse tree returned by step 1, visit each node, performing the required restructuring to the parse tree in order to convert the SQL query into the equivalent multi-level SQL query. Depending upon the query, different types of restructuring are performed. SQL statements such as CREATE, GRANT and UPDATE require creating several parse trees, while other SQL statements only require the removal of some nodes and/or node augmenta-

tion. Regardless of the type of query, the parse tree representing that query is first fully expanded to ensure no loss of information, then reduced when required. Reduction is required when tuple level classification is expressed in the query and the table referred to by the tuple level classification is a view that is going to be converted to a base relation. The tuple level classification is dropped, because base relations do not have tuple level classification attributes. Tuple level classification attributes are associated with views. This step is referred to as the restructuring phase.

4. Starting with the root of the first restructured parse tree within the forest returned by step 3 and continuing until each tree in the forest has been visited, convert the parse tree(s) into an SQL statement. At this point the original query entered by the user has been translated into its equivalent multi-level version. The multilevel version is sent to standard output. This step is referred to as the output generation phase.

5. All parse trees created in the above steps are deleted and the preprocessor returns to step 1.

5.2 Advantages of using C++

We used C++ instead of a language that didn't support the object oriented methodology, because through the use of polymorphism, dynamic binding and inheritance, we were able to write code that is modular and extensible. For example, every SQL statement must be restructured as it is being parsed. Using a conventional language such as C, one might implement the restructuring phase as follows:

```
restructure (stmt_type) {
   switch (stmt_type) {
   case UPDATE:
      /* perform update restructuring */
   case SELECT:
      /* perform select restructuring */
   case DELETE:
      /* perform delete restructuring */
   .
   .
   .
   }
}
```

The disadvantage of this approach is that the restructure function must know about every SQL statement and each time a new statement is added, the restructure function must be updated. Using C++ one could write the above as,

```
class statement   : public parse_node {
protected:
      union {
        class upd_srch_st_nd    *updsst;
        class del_srch_st_nd    *dsrchst;
        class sel_st_nd         *selst;
                .
                .
                .
      };
public:
   void restructure();
};

/* restructure method for class statement */

void statement::restructure()
{
   ((parse_node*)updsst)->restructure(vslev);
}
```

and define a restructuring method for each of the classes upd_srch_st_nd, del_srch_st_nd, and sel_st_nd. When the system was compiled, the polymorphic features of C++ would enable updsst to reference any of the types listed in the union above and dynamic binding would select the appropriate restructure method at run-time. If the generic restructure method defined in class parse_node was sufficient, then that method would get called via inheritance. When the time arrived to add a new SQL statement, only the interface to class statement would need updating. The restructure method defined for statement would not need changing and could be kept in a library. At run-time, it would automatically know about the new statements.

5.3 Communication with the Global Scheduler

The preprocessor is invoked by the global scheduler through the use of the C++ type `parser_starter`. See the *Design Documentation for the SINTRA Global Scheduler* [KaP93] document for more information about the `parser_starter` type. Once the preprocessor has been started, it reads from standard input until a semi-colon is received, all the characters read before the semicolon are passed through the algorithm described in section 5.1. This process continues until the EXIT string is received, at which point the preprocessor terminates.

6 Summary

In this document, we have described functions and implementation details of the preprocessor of the SINTRA prototype. In conjunction with the companion documents, [Kan94] and [CKF94], this document should, we hope, provide valuable insights to the designers of the next generation preprocessors.

References

[BeL76] Bell, D. E., and LaPadula, L. J. Secure computer systems: Unified exposition and multics interpretation. The Mitre Corp, (1976).

[FrM89] Froscher, J. N., and Meadows, C. Achieving a trusted database management systems using parallelism. in Database Security II: Status and Prospects (North-Holland 1989)

[Kang] Kang, M. H. Optimization techniques for a multilevel database systems. In preparation.

[Kan94] Kang, M. H., *et. el.* Achieving Database Security through Data Replication: The SINTRA Prototype. To appear in Proceedings of the 17th National Computer Security Conference (1994).

[KaP93a] Kang, M. H., and Peyton, R. Design documentation for the SINTRA global scheduler. Naval Research Laboratory Memo Report 5542-93-7362 (1993).

[CKF94] Costich, O., Kang, M. H., and Froscher, J. N. The SINTRA Data Model: Structure and Operations. To appear in Proceedings of the IFIP 8th Working Conference on Database Security (1994).

[KFC93] Kang, M. H., Froscher, J. N., and Costich, O. A practical transaction model and untrusted transaction manager for multilevel-secure database systems. Database Security VI: Status and Prospects (North-Holland 1993).